计算机网络

1. HTTP--超文本传输协议

是应用层上的一种客户端/服务端模型的通信协议，由请求和响应构成，无状态

协议：规定了双方按照约定格式才能准确的通信

无状态：两次通信间没关系

Socket 是：TCPAPI，为了方便使用 TCP/UDP 抽象出来的一层，是应　　用层和传输控制层之间的一组**接口**，用于组织数据去符合指定协议

WebSocket：一个 HTTP **应用层协议**--只需要一次握手然后单独建立一条 TCP 通信通道来传数据(用于传小数据不占很多宽带)

OSI 7 层模型 用处：规范地控制网络：

HTTP /FTP 文件传输协议/webSocket　TCP/UDP　IP　数据链路　物理介质

应用层 表示层 会话层 传输层 网络层 数据链路层 物理层

TCP/IP 5 层模型及功能：

1.物理层：将数据以实体呈现并传输

2.数据链路层：把数据分割成特定的可被物理层传输的帧

3.网络层：**选择传送路径**

4.传输层：给**数据编号**、控制数据流量、查错与错误处理，确保数据可靠、**按顺序、无错地传输**

5.应用层：负责在网络中的**两节点之间建立和维持通信**。文件传输

为什么分层：1.各层独立，不用知道下**层怎么实现**，调整层内工作方便

2.不会有**木桶效应**，不会因某一层**技术不完善**影响整体效率

3.易于**维护**，对每层单独调试即可

HTTP 状态码：80 未加密 明文传输 100—199 服务器收到要请求者继续执行请求的状态代码 200—299 成功处理了请求的状态代码 300-399 重定向（完成请求要其他操作） 400-499 客户端出错 500—599 服务端出错

常见状态码：100 要客户端继续发

200 服务器接收到请求，将返回结果

202 接受了请求，不确定处不处理

301 请求的链接发生永久性变化，客户端向新链接请求

302 请求的链接发生临时性变化 临时重定向

304 get 请求被运行但请求的网页没修改

400 请求无效

401 当前用户需要验证

403 服务器得到请求但是拒绝执行

404 请求失败，请求的资源没找到或者不存在

URL 构成

HTTP **请求报文**：请求行(请求方法,URL,版本)，请求头(content-Type 比如 image/gif 是 gif 图片格式)，空白行(通知服务器不再有请求头)，请求体(POST 的数据)

HTTP **响应报文**：响应行(版本，状态码，状态码描述)，响应头
空白行，响应体(html)

常用请求头：**Accept**(可接受响应内容类型), cache-control(指定当前请求/回复**是否用缓存**机制)

实现长连接：创建**请求方法**-设置**请求地址和参数编码格式(utf8)**-设置**长连接**-设置请**求参数**-执行

端口号：

IP 区分计算机，**端口号区分相同计算机的不同的服务，**一台计算机上可以同时提供很多个服务，如数据库服务、Web 服务。

客户端用 IP 来找服务端，用端口号来找服务端的服务，实现真正的访问

缓存机制：第一次向服务器发请求，服务器返回请求的资源，之后再要请求资源就可以用**强缓存**(200，cache-control)(从本地缓存获取数据，**不**与服务器**交互**)或**协商缓存**(304，Last-Modified)(发请求到服务器，服务器判断可不可以用本地缓存)

短连接：客户端和服务端每次 **HTTP 操作就建立一次链接，任务结束就断开，**用于 web 网站的 http 服务因为会有很多客户端连接

长连接：连接后客户端或服务端关闭，客户端再次访问时继续用这个 TCP 连接
用于操作频繁(数据库)，点对点的通讯，而且连接数要少，因为会消耗资源

HTTP 0.9 只能发 GET 请求

HTTP 1.0 支持**短连接**，而且会**请求部分对象**，但服务器把**整个对象**送过来了

HTTP 1.1 支持**长连接**，加了 **cache-control 请求头**，**宽带优化**(只送请求部分对象)

HTTP2.0:基于 **https**，具有**安全性**，使用**二进制格式**和**多路复用**(1.1 里只能完成上个请求才能发起下个请求，而 2.0 里的二进制帧可交错传输)

2. TCP 和 UDP 的区别

是传输层两个协议

1. 区别

TCP 面向连接，　　UDP 面向无连接

TCP 开销小，有拥塞控制　UDP 没有不管网络是否拥塞，客户端都能一直发

TCP 面向字节流　　UDP 基于数据包

TCP 保证数据正确性和顺序，UDP 可能丢包而且不保证顺序

组成：

UDP 很简单，除了**端口号**几乎什么都没有，**不会根据网络的情况进行拥塞控制**

变可靠: 要**接受方收到 UDP 后回复个确认包**，发送方收不到确认包就要重新发

TCP 除了有源端口和目标**端口**，还有包的**序号**(给包编号知道哪个先来)，确认序号(确认对

方是否收到，解决丢包)，**状态位**(SYN 发起链接，ACK 回复,RST 重新连接，FIN 结束连接)
流量控制：
滑动窗口作用：用来**加速数据传输，**TCP 要保证“可靠”，就需要对一个数据包进行 ack 确认表示接收端收到。有了滑动窗口，接收端就可以等收到许多包后只发一个 ack 包，确认之前已经收到过的多个数据包。有了滑动窗口，发送端在发送完一个数据包后不用等待它的 ack，在滑动窗口大小内可以继续发送其他数据包
TCP **滑动窗口：窗口大小指一次传输几个数据**，发送方在发送时候始终保持一个窗口，只有窗口内的数据帧才被发送，同样接收方也有个接收窗口，只有在窗口内的才会被接收，通过改变窗口和窗口大小实现流量控制
窗口太大：容易**丢包**->重发->耗费带宽 **窗口太小：频繁**发->占**带宽**
拥塞控制(防止发送方发的太快导致过多数据注入到网络—死锁)发送端发报文速度比接收端接受速度快　拥塞：资源需求 > 可用资源
发送方有一个拥塞窗口变量，越拥塞窗口越小
知道丢包：1.定时器超时 2.收到 3 个重复的 ACK
什么时候拥塞控制：1.网络传输 TCP 报文过程中发生丢失报文 2.报文需要重传
4 个拥塞机制：
慢开始：由小到大增加拥塞窗口数值
拥塞避免：按线性增长拥塞窗口，比慢开始慢
重传定时器溢出的时候(也就是有很多报文没有按时发送到接收端或者接收端的 ACK 报文没有到达发送端)用慢开始和拥塞避免 先慢开始(指数增长)再拥塞避免(线性)
快重传：接收方收到一个失序报文段立即发出重复确认，而不是自己发数据时捎带确认
快回复：使慢开始在 TCP 连接建立和网络超时时采用
报文重传了
拥塞控制 vs 流量控制：流量控制虽然可以高效可靠的传送大量的数据，但是如果在刚开始阶段就发送大量的数据，可能会导致网络拥堵，因为网络上的计算机太多了

1.相同点

> (1) 现象都是丢包;
> (2) 实现机制都是让发送方发的慢一点，发的少一点

2.不同点

> **(1) 丢包位置不同**
> 流量控制丢包位置是在接收端上
> 拥塞控制丢包位置是在路由器上
> **(2) 作用的对象不同**
> 流量控制的对象是接收方，怕发送方发的太快，使得接收方来不及处理
> 拥塞控制的对象是网络，怕发送发发的太快，造成网络拥塞，使得网络来不及处理

3.联系

```
1  拥塞控制
2      拥塞控制通常表示的是一个全局性的过程，它会涉及到网络中所有的主机、
3      所有的路由器和降低网络传输性能的所有因素
4  流量控制
5      流量控制发生在发送端和接收端之间，只是点到点之间的控制
```

应用：

UDP：DNS 协议，看视频, 发语音, QQ 聊天, 共享屏幕 因为效率高但准确率低
TCP：HTTP 协议，QQ 传文件，邮件，登录 因为准确率高效率低

TCP 粘包：发送方发送的几个包数据到达接收方时粘成了一包，从接收缓冲区来看，后一包数据的头紧接着前一包数据的尾

原因：发送方用 Nagle 算法—收集多个小分组，在一个确认到来时一起发送，接收方收到数据包保存到缓存里而不是直接给应用层处理，然后应用程序再到缓存读取包。如果包到缓存的速度大于应用程序从缓存读取包的速度就会造成多个包被缓存，应用程序读到多个首尾相连在一起的包。

解决：发送方关闭 Nagle 算法，应用层循环读取数据，但每条数据要有固定格式而且发送方要把数据长度一起发送，这样就能判断开始和结束的位置

UDP 不会粘包因为是面向消息传输的，有保护消息边界，接收方只接受一条独立地信息

TCP 是基于流传输的，不认为消息是一条一条的，没有保护消息边界

3. 代理

proxy 代表访问用户时是代理，比如我们访问一个网址，不希望该网站知道我们的 ip，就找了一个 proxy，网站以为 proxy 的 ip 就是我们的 ip。相反，proxy 代表被访问的服务器，则此时 proxy 是反向代理，web-server 希望对用户屏蔽一些信息就找了一个 proxy 隔在中间，此时 proxy 代表 web-server 集群，用户以为 proxy 的 ip 就是就是被访问 web-server 的 ip，反向代理的服务器要有**负载均衡**功能（在多个资源中分配负载，达到最优化资源使用，避免过载）。使用服务组件 Nginx 实现。四层负载均衡主要工作于传输层，主要是处理消息的传递而不是内容，七层负载均衡主要工作于应用层处理消息内容。七层比四层优势：可以终止网络传输，根据消息**内容**作出负载均衡决定，CPU 更密集，劣势：要为每一种应用服务专门开发一个反向代理服务器，这样限制了反向代理负载均衡的应用范围

4. TCP 三次握手

三次握手的本质是确认通信双方收发数据的能力。

第一次客户端生成一个序列号给服务端发连接请求，报文段包含了序列号和 SYN 标志位=1。服务端知道客户端发件和他自己收件能力。

第二次服务端看到 SYN=1 是个连接请求，存下客户端序列号，之后随机成一个序列号，回复的报文里有标志位 SYN=ACK=1，新序列号，以及 ack=客户端序列端+1。客户端收到了，客户端知道客户端发件收件和服务端的发件收件能力 OK。

第三次客户端收到 SYN=1 说明同意连接，存下服务端序列号，发送的报文里有

ACK=1，ack=服务端序列号+1，序列号=第一次握手序列号+1。服务端知道服务端发件和客户端收件能力 OK

为什么不是 2 次握手？

因为要考虑连接时丢包的问题，如果只握手 2 次，第二次握手时如果服务端发给客户端的确认报文段丢失，此时服务端已经准备好了收发数据(可以理解服务端已经连接成功)，而客户端一直没收到服务端的确认报文，所以客户端就不知道服务端是否已经准备好了(可以理解为客户端未连接成功)，这种情况下客户端不会给服务端发数据，也会忽略服务端发过来的数据。如果有三次握手客户端发的确认 ack 报文丢失，服务端在一段时间内没有收到

确认 ack 报文的话就会重新进行第二次握手，也就是服务端会重发 SYN 报文段，客户端收到重发的报文段后会再次给服务端发送确认 ack 报文。

5.四次挥手

四次挥手目的是关闭一个连接。比如客户端初始化的序列号 ISA=100，服务端初始化的序列号 ISA=300。TCP 连接成功后客户端总共发送了 1000 个字节的数据，服务端在客户端发 FIN 报文前总共回复了 2000 个字节的数据。

第一次挥手客户端发出连接释放，报文里有 FIN 标志位=1，序列号=1101（1 建立连接时占用得序列号）。之后客户端不能发数据只能收数据。

第二次挥手服务端收到 FIN 报文回复确认报文，里面有标志位 ACK=1，确认号 ack=1102（FIN 报文序列号+1），序列号为 2300（起始的加回复的字节数据）服务端可能还有数据没发完要处于关闭等待状态一会儿，不会立马发 FIN 报文。

第三次服务端把最后数据比如 50 字节发完了，就发送释放报文，报文里 FIN=ACK=1，ack=1102 和第二次一样，序列号=2300+50=2350。

第四次挥手客户端发确认报文，ACK=1，ack=2351，序列号=1101+1

为什么 TCP 连接有 3 次关闭时候有 4 次？

因为只有客户端和服务端都没有数据发送时才能断开 TCP。而客户端发 FIN 报文只能保证客户端没数据要发，服务端还有没有数据要发不知道。所以服务端收到客户端的 FIN 报文要先发一个确认报文等数据发完了再发 FIN 报文。所以这个不能一次性把确认报文和 FIN 报文发给客户端就多了一次

为什么客户端发出第四次挥手确认报文要 2MSL（最长报文段时长的 2 倍）之后才能释放 TCP 连接？/ Time_wait()的作用

这样是考虑丢包问题，如果**第四次报文丢失**，服务端没收到确认号 ack 报文要重新发第三次挥手报文，这样报文一去一回时间正好是 2MSL。所以需要这么长时间确认服务端收到

6.浏览器输入 URL 会输出什么

1.浏览器**向 DNS 发送域名**，获取 **IP**。先检查浏览器**缓存**有没有这个**网址映射**，有就调用，没有则去**操作系统的缓存**里找，没有则去找 **LDNS** 有没有，如**果 LDNS 仍然没有命中，就直接跳到 Root Server 域名服务器请求解析**

5. 根域名服务器返回给 LDNS 一个所查询域的主域名服务器（gTLD Server，国际顶尖域名服务器，如.com .cn .org 等）地址

6. 此时 LDNS 再发送请求给上一步返回的 gTLD

7. 接受请求的 gTLD 查找并返回这个域名对应的 Name Server 的地址，这个 Name Server 就是网站注册的域名服务器

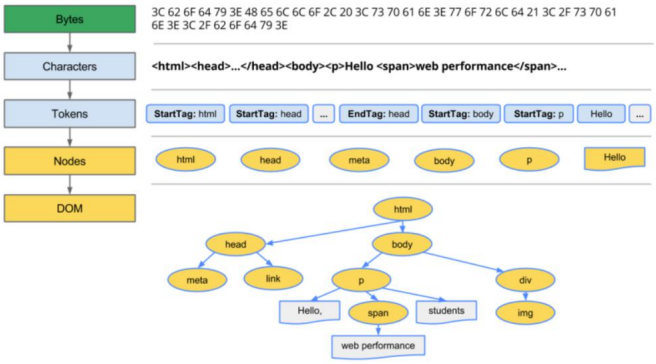
8. Name Server 根据映射关系表找到目标 ip，返回给 LDNS

9. LDNS 缓存这个域名和对应的 ip

10. LDNS 把解析的结果返回给用户，用户根据 TTL 值缓存到本地系统缓存中，域名解析过程至此结束

2.得到 **IP 和端口号 80or443 之后**，会调用系统库函数 **socket**，请求一个 **TCP 流套接字**，**客户端向服务器发 HTTP 请求报文**：应用层：应用层发请求报文, 传输层：三次握手建立连续, 网络层：路由寻址,数据链路层：传输数据, 物理层：物理传输 bit

3.服务器**解析请求报文，发响应报文**
4.TCP **四次挥手**，关闭连接
5.客户端**解析响应报文**，浏览器开始**显示 HTML**，js 文件开始建立 dom 树
CSS 解析器解析 css 文件得到 CSS 树，浏览器结合 CSS 数和 dom 树(下载图片等)一起建立 render 树，这里主要是排除非视觉节点，最后渲染 render 树
构建 DOM 树：



7.GET 和 POST 的区别--两种发送请求方法

HTTP 最早被用来做浏览器与服务器之间交互 HTML 和表单的**通讯协议**；后来又被广泛的扩充到**接口格式的定义**上。底层是 TCP/IP
1. 用途和幂等性
浏览器的 GET 用于获得 HTML 页面，图片，css，js 等资源。
只是读取数据 -> 可以对数据做缓存（做到浏览器 代理 服务端都可以）->幂等(副作用相同) 幂等就是多次操作副作用相同
POST 用于更新资源信息 ->不是幂等(副作用不同)->不能多次执行和不能缓存。
比如一个下单页面 ->下单成功被存到缓存，下次下单就不会对服务器发送请求 ->直接返回缓存的下单成功页,导致没有下单。当然 get 也可以有副作用 post 也可以没有。
2.数据位置
GET 的数据附在 **URL 之后的 queryString 传输**，更加容易被看到，更不安全
POST：请求的 **body** 传输数据，安全一些（但是 HTTP 实现**接口**发送请求时（ajax，postman 发出请求）参数可以放任何位置比如 header body querystring..这种不确定性会造成低效所以用了 REST 规定了增删改查)

RESTFUL 规范：

1.url 链接设计：采用 https 方式，有 api 关键字，有版本需要明确版本，请求链接用名词来表示资源，具体的操作方式采用请求方式来确定

2.url 响应数据设计：需要明确 状态码、错误信息、成功结果，子资源一般用子资源的接口来标注

但是私密数据在 body 里也是会被记录的，所以都不安全，所以要通过 https
3.数据大小
GET 方法传输的数据量是 **2KB** 而且在 URL 种传送的参数有**长度限制**，POST 都没有，但真正影响还是**服务器和 HTTP 的规定**
4.**请求缓存**时

GET 请求类似于**查找**过程，用户获取数据，**不用每次和数据库连接**，POST 一般是修改和删除的工作, **必须和数据库交互** -> GET 适合请求缓存
5.GET 产生一个 TCP 数据包，POST 产生 2 个:
GET 请求，浏览器会把 http header 和 data 一起发送，服务器响应 200，
POST 请求，先发 http header,服务器响应 100continue，再发 data，服务器响应 200
但是 GET 和 POST 本质就是 **TCP 链接**，并无差别，是 HTTP 的规定和浏览器的限制导致他们不同

8.http 请求方法

GET 请求**页面信息**
POST 向指定资源**提交数据进行处理**
PUT 传送的**数据取代**指定**内容**
DELETE **删除**指定**页面**
OPTIONS 允许客户端查**看服务器性能**
HEAD 类似 GET，但返回的**响应没有内容只有报头**
TRACE **显示**服务器收到的**请求**，用于测试或诊断

9.http 和 https 的区别？

http 端口号 80，运行于 TCP 之上，**明文传输**，客户端和服务端**无法验证**对方**身　份，资源消耗少，安全性低**
https 端口 443，HTTP 协议+SSL 协议，是添加了加密和认证的 http，加密通过对称加密和非对称加密混合加密 -> 消耗更多 CPU 和内存，需要花钱买**证书**(分 DV, OV, EV 三种类型可信度越来越高，越来越严格)
为什么混合加密：
数据是被对称加密传输的，对称加密过程需要客户端的一个密钥，为了确保能把该密钥安全传输到服务器端，采用非对称加密对该密钥进行加密传输，总的来说，对数据进行对称加密，对称加密所要使用的密钥通过非对称加密传输。
过程：1.客户端向服务器发起 HTTPS 请求，连接到服务器的 443 端口
2.服务器端有一个密钥对，即公钥和私钥，是用来进行非对称加密使用的，服务器端保存着私钥，不能将其泄露，公钥可以发送给任何人。
3.服务器将自己的公钥发送给客户端。
4.客户端收到服务器端的证书之后，会对证书进行检查(看签发机构可不可信任)，如果发现证书有问题，那么 HTTPS 传输就无法继续。如果公钥合格，那么客户端会生成一个用于进行对称加密的密钥。然后用服务器的公钥对客户端密钥进行非对称加密，这样客户端密钥就变成密文了，至此，HTTPS 中的第一次 HTTP 请求结束。
5.客户端会发起 HTTPS 中的第二个 HTTP 请求，将加密之后的客户端密钥发送给服务器。
6.服务器接收到客户端发来的密文之后，会用自己的私钥对其进行非对称解密，解密之后的明文就是客户端密钥，然后用客户端密钥对数据进行对称加密，这样数据就变成了密文。
7.然后服务器将加密后的密文发送给客户端。
8.客户端收到服务器发送来的密文，用客户端密钥对其进行对称解密，得到服务器发送的数据。第二个 HTTP 请求结束

对称加密加密解密用的同一个密钥，传输过程会被截获，不安全，但传输快
其加密过程如下：**明文 + 加密算法 + 私钥 => 密文**
解密过程如下：　**密文 + 解密算法 + 私钥 => 明文**
由于对称加密的算法是公开的，所以一旦私钥被泄露，那么密文就很容易被破解，所以对称加密的缺点是密钥安全管理困难。
非对称加密用公钥加密，私钥解密，公钥无法反推导出私钥所以安全，但加密解密废时间，速度慢
被公钥加密过的密文只能被私钥解密，过程如下：
明文 + 加密算法 + 公钥 => 密文，　密文 + 解密算法 + 私钥 => 明文
被私钥加密过的密文只能被公钥解密，过程如下：
明文 + 加密算法 + 私钥 => 密文，　密文 + 解密算法 + 公钥 => 明文
SSL 是安全套接层协议 TLS 安全传输层协议

10.Session 和 Cookie

Cookie 是服务器保存在浏览器上的 key-value 格式文件，有用户信息
　　最大作用：存储 sessionId 来唯一标识用户
Session 依赖 Cookie 实现, 是**服务器分配的一块储存空间**
过程：客户端向服务器发请求，如果**服务器需要记录该用户状态**，就颁发浏览器一个 Cookie，并且自己创建一个 session -> **客户端**浏览会把 Cookie **存**起来，再请求该网站时，浏览器会把请求网址和 Cookie 一起发给服务器。服务器检查该 Cookie 来里的 sessionID 来辨认用户身份。

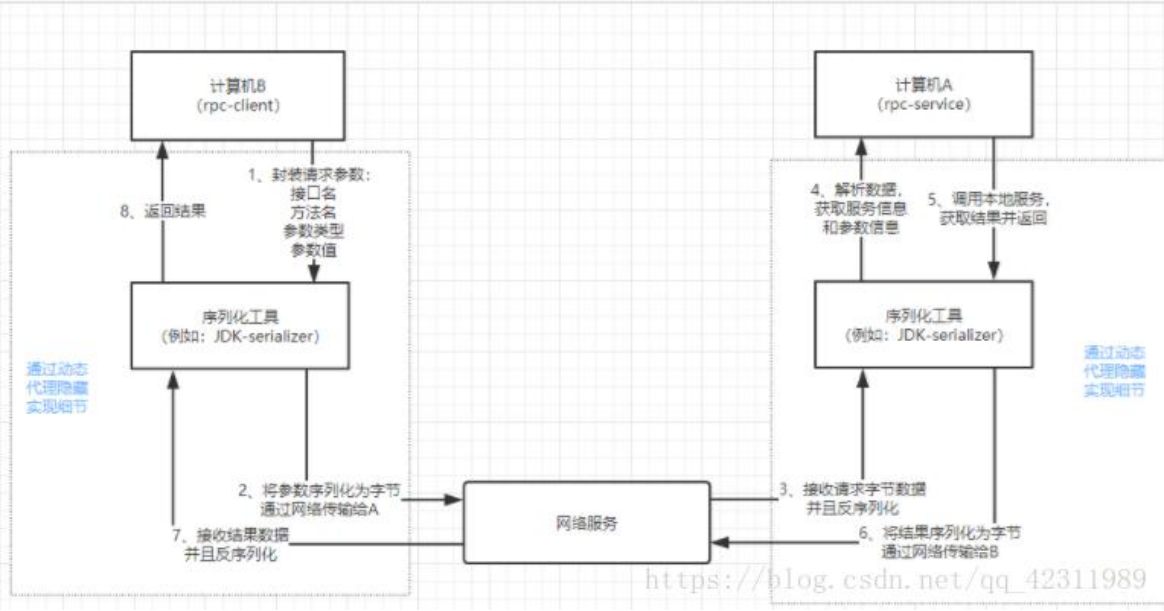
cookie、session(放敏感信息)与 localStorage 区别

1. 位置
cookie 数据在客户端，安全性低　session 数据在服务器，安全性高
2. 大小
Session 和 localStorage 比 cookie 大得多
3. 存在时间
　localStorage 存储持久数据，只能手动删
　sessionstorage 关浏览器
　cookie 设置了过期时间，浏览器关了也不会删，没设置则关闭浏览器后失效
　session 有失效时间，如果 session 倍访问，失效时间归 0 开始计

token
cookie 和 token 都是**首次登录**时由服务器**下发用来验证**
好处：不用账号密码 -> 减少数据库查询
区别(管理、共享)：token 由**应用管理**, 可以**避开 csrf** 攻击并能在多个**服务间共享**
csrf：浏览器访问服务器后得到 cookie，之后访问非法网站之后，cookie 和里面的 ticket(凭证)被盗取。之后非法网站模拟你的身份用 post 表单向服务器提交数据。
如果提交的是转账的数据就很危险有可能将钱划走
Spring security 解决方案：会在返回给浏览器的表单里添加一个 tocken，tocken 每次请求都不一样，是随机生成的，非法网站能窃取 cookie 但不能猜到 tocken，所以 post 提交数据

的时候，服务器看到 ticket 对但没有 tocken 则判断为 csrf 攻击，提交失败（但 security 只能在非异步请求，在返回网页时自动传一个 tocken，异步时需要咱们自己处理，前端模板上编写逻辑加入 tocken ）

Unicode 16 进制 UTF-8 二进制 UTF-8 是 Unicode 的一种实现
RST 标志位表示复位，用来异常的关闭连接。用 RST 包关闭连接时不用等缓冲区的包都发出去（不像 FIN），直接丢弃缓冲区的包发送 RST 包，接收端收到 RST 包后也不用发 ACK 包确认。比如 A 向 B 连接，但 B 没有监听相应地端口，B 的 TCP 处理程序就会发 RST 包
RST 攻击：AB 建立了连接，C 伪造一个 TCP 包发给 B，让 B 丢弃缓冲区的数据而且异常地和 A 断开连接。想伪造 A 的包要知道 A 的源 IP、源端口、目标 IP、目标端口
RPC 是远程过程调用，也就是在一个服务器的应用要调用另一个服务器上的应用的方法，但不在一个内存空间，不能直接调用，只能靠网络来调用



Linux 指令

scp，管道传输：Linux 主机之间传输文件
ps -A：显示**所有**程序
ps -ef：用**标准格式**显示进程
ps aux:用 BSD 的**格式**显示进程
file：**判断**文件类型
fd：查看已被打开的文件描述符
grep walkpgdir 文件名：print 出文件中**匹配行内容**
grep -i :不区分大小写 print 出文件中匹配行
find：在目录重搜索文件，有则 print 出文件名
top：CPU 使用状态，内存使用情况
vi 替换字符串：s/vivian/sky 替换**当前**行**第一个** vivian 为 sky

s/vivian/sky/**g** 替换**当前**行**所有** vivian 为 sky
%s/vivian/sky/g 替换**所有行**所有 vivian 为 sky
more **分页展示**文件内容
less **分屏展示**内容，根据**显示需要加载内容**，对**大型文件**效率更高
head -n 5 查看文件头 5 行， tail 反之。
ifconfig:显示网络接口，子网掩码
netstat：看网络情况
netstat -anpt | grep 80：查看 80 端口 – 与网络有关用 netstat 再用 grep 找端口
Lsof -i:端口号 ：查看端口被什么进程占用
ps aux | grep sha(程序的名字)：查线程号
ps -ef |grep mysql 显示有关 mysql 的进程
查看日志：tail/head 来显示 test.log 内容 + cat -n test.log | grep “debug”查询关键字的日志
vim 进入 log 文件用/+关键字查找，下一个按 n 就行
kill -9 pid：杀死进程 + 重新加载和停止进程，发信息 9 是 SIGKILL(发给进程)

Git 指令

git branch 查看本地所有分支
git rm 文件名 删除指定文件
git log 查看 commit 日志
git clone 克隆版本库
把本地文件/文件夹 移到 github 上：
git config --global user.name “” --设置 github 用户信息
git config --global user.email “”
git init 初始化 --> 生成一个 .git 文件夹
git add . 添加当前文件夹下所有文件
git commit -m “”提交到 Repository
git remote add origin 加上仓库地址
git push origin master 将文件推到服务器上
merge 和 rebase 用于合并分支
区别：1.merge **不会**保留 merge 分支的 **commit 到日志**
2.**处理冲突(不知道要保留什么设置)**时，merge **会产生一个 commit**，rebase 不会
版本回退：
1.没用 git add.缓存代码：git checkout — filepathname
2.用了 git add：用 git reset HEAD filepathname
3.已经提交了代码：git log 看 git 提交历史，里面第一行就是 commitid，然后用这个 id 去 git reset —hard commitid 回退
4.已经 push 到远程服务器：git reset —hard
git push origin HEAD —force

版本控制：是记录软件开发过程里文件内容的变化用来查特定版本修订情况的系统

好处：

1. 记录什么人什么时候改了什么内容，每一次文件的改变，文件的版本号都将增加，确保在开发过程中由不同人所编辑的内容都得到更新
2. 方便并行开发，因为可以有效地解决版本的同步以及不同开发者之间的开发通信问题，并行开发中最常见的不同版本软件的(Bug 修正问题也可以通过版本控制中分支和合并的方法解决

版本控制系统：

Git(分布式) 客户端并不只提取最新版本的文件快照，而是把原始的代码仓库完整地镜像下来。这么的话，任何一个服务器发生故障，事后都可以用任何一个镜像出来的本地仓库恢复

SVN/ CVS(集中式) 一个单一的集中管理的中央服务器，它保存了所有文件的修订版本，然后一起工作的人都可以通过客户端连到这台服务器，拿到最新的文件或者提交更新

Python(一切都可看做对象)

1. 去首尾空格：strip，去掉左空格：lstrip，去掉右空格：rstrip
2. list：有序集合，靠索引增删 insert，pop(index),append 和 pop()是对 list 末尾增减。
3. tuple：有序列表，只能用[]获取元素，只能用[][]改 tuple 里的 list 的元素
4. 切片 startindex：endindex：step，
   start 省略时，step 为正则从起点开始，为负 从终点开始
   end 省略时，step 为正时取到终点，为负时取到起点
   取偶数位[::2]取奇数位[1::2]

5. 浅拷贝：只拷贝数据集合最外边的一层，深层的数据只是做了内存地址引用，并没有拷贝。假如 a 是一个包含数组的数组，b 浅拷贝了 a，则 a 改了他的数组元素里的数组元素，b 也会相应地改。

深拷贝：完全拷贝数据集合的所有数据，与源数据再无相关。

浅拷贝

- 只拷贝数据集合最外边的一层，深层的数据只是做了内存地址引用，并没有拷贝。

例如一个列表里面嵌套了另一个列表：[1, 2, 3, [4, 5]]

```
>>>import copy                # 导入模块
>>>a = [1, 2, 3, [4, 5]]
>>>b = copy.copy(a)           # 使用浅拷贝拷贝
>>>b                          # 打印输出b
[1, 2, 3, [4, 5]]
>>>id(a),id(b)                # 打印输出a、b
(2692110485192, 2692110468616)
>>>a is b                     # a和b的内存地
False
>>>a[3] is idb[3]             # 内层的b的[4,
True

>>>a[3][1] = 10               # 更改a的内层列
>>>a                          # 打印输出a
[1, 2, 3, [4, 10]]
>>>b                          # 打印输出b，可
[1, 2, 3, [4, 10]]
```

6. 常用库：numpy(科学计算包)　beautifulsoap(爬虫相关)
   requests (HTTP 请求库)　　pip(安装和管理 python 包的工具)
7. 在运行时才确定对象的类型和内存--动态类型　(不像 Java，C++)
8. 内存管理：用一个私有 Heap 来放所有对象和数据结构，我们无法访问
   引用计数：x=3.14,x 是第一个引用,y=x，y 是第二个引用，计数=2
   垃圾回收：清楚引用计数=0 的对象的内存和互相引用的两个对象
   用 id 得到对象身份标识，is 比较对象地址
9. 数据类型：不可变：Number，String，tuple
   可变：List，dictionary，set
   a=1 b=a b+=1 a 还是 1
   不可变是指 b=a 会分配 b 一个新的内存，并把 a 的值复制给 b
   可变指=后两个对象指向同一个内存
10. 内存管理：创建大量消耗小内存的对象，频繁调用 malloc 会导致很多内存碎片，减低效率，内存池就是先在内存申请一定数量、大小相等的内存块备用，有内存需求先从内存池分配，不够再申请，这样就能减少内存碎片，提升效率

小于 256bits，pymalloc 申请内存
大于 256bits，new/alloc 申请内存
回收：对象引用计数=0，从内存池来的到内存池去

读文件的过程 1、进程调用库函数向内核发起读文件请求；
2、内核通过检查进程的文件描述符定位到虚拟文件系统的已打开文件列表表项；
3、调用该文件可用的系统调用函数 read()
3、read()函数通过文件表项链接到目录项模块，根据传入的文件路径，在目录项模块中检索，找到该文件的 inode；
4、在 inode 中，通过文件内容偏移量计算出要读取的页；
5、通过 inode 找到文件对应的 address_space；
6、在 address_space 中访问该文件的页缓存树，查找对应的页缓存结点：
1.如果页缓存命中，那么直接返回文件内容；
2.如果页缓存缺失，那么产生一个页缺失异常，创建一个页缓存页，同时通过 inode 找到文件该页的磁盘地址，读取相应的页填充该缓存页；重新进行第 6 步查找页缓存

写文件

前 5 步和读文件一致，在 address_space 中查询对应页的页缓存是否存在：
6、如果页缓存命中，直接把文件内容修改更新在页缓存的页中。写文件就结束了。这时候文件修改位于页缓存，并没有写回到磁盘文件中去。
7、如果页缓存缺失，那么产生一个页缺失异常，创建一个页缓存页，同时通过 inode 找到文件该页的磁盘地址，读取相应的页填充该缓存页。此时缓存页命中，进行第 6 步。
8、一个页缓存中的页如果被修改，那么会被标记成脏页。脏页需要写回到磁盘中的文件块。有两种方式可以把脏页写回磁盘：
1. 手动调用 sync()或者 fsync()系统调用把脏页写回
2. pdflush 进程会定时把脏页写回到磁盘
同时注意，脏页不能被置换出内存，如果脏页正在被写回，那么会被设置写回标记，这时候该页就被上锁，其他写请求被阻塞直到锁释放